Aller Anfang ist schwer! Wie gelingt der Einstieg in den Informatikunterricht?

Dieter Engbring[1]

Abstract: Dieser Aufsatz befasst sich mit dem Umstand, dass nicht allen SchülerInnen mit der erwünschten Geschwindigkeit und den erwarteten Erfolgen der Einstieg in den Informatikunterricht gelingt. Zu den Gründen werden in diesem Aufsatz Hypothesen vorgelegt, die sich aus Beobachtungen im Informatikunterricht sowie aus Gesprächen mit LehrerInnen und SchülerInnen ergeben. Zusätzlich werden alternative Vorgehensweisen vorgestellt, die nicht nur einen anderen Informatikanfangsunterricht beschreiben, sowie der Überprüfung der Hypothesen dienen.

Keywords: Anfangsunterricht, Programmierumgebungen und -sprachen, Objektorientierung

1 Einleitung

Nicht nur hinter vorgehaltener Hand, sondern ganz offen reden Informatik-LehrerInnen darüber, wie schwer es einigen ihrer SchülerInnen fällt, selbst die einfachen und grundlegenden Begriffe zu lernen und leichte Aufgaben zu lösen. In der fachdidaktischen Literatur gibt es bislang keine Untersuchungen, die den Gründen und dem Ausmaß dieser Befunde nachgehen. Auch dieser Aufsatz kann hierzu allenfalls erste Hypothesen liefern, aus denen dann eine systematische Untersuchung entwickelt werden soll (2). Diese Hypothesen werden durch einen Alternativvorschlag für den Einstieg (3) und einem Forschungsansatz zur Überprüfung der Hypothesen und zur Evaluation des alternativen Zugangs ergänzt (4). In einem Fazit mit Ausblick wird auf die jetzt anstehenden Forschungsarbeiten hingewiesen (5).

2 Befunde zum Informatikunterricht in der Einführungsphase

Das Implementieren und damit auch die Programmiersprache sowie die Entwicklungsumgebungen spielen eine große Rolle in der alltäglichen Praxis des Informatikunterrichts in der Einführungsphase der gymnasialen Oberstufe, die immer noch für viele SchülerInnen der Informatik-Anfangsunterricht ist. Zwar ist es das weiterreichende Ziel, die dahinterstehenden Techniken und Denkweisen der informatischen Modellierung zu vermitteln, die auch – siehe Computational Thinking [Wi06] oder ähnliche Ansätze – über die Informatik hinaus anwendbar sind. Ansätze, diese Kompetenzen ohne Implementierung, frei von einer Programmiersprache und einer dedizierten ggf. auch einer für unterrichtliche Zwecke gestaltete (Entwicklungs-)Umgebung zu erreichen, greifen allerdings wohl

[1] Institut für Informatik, Universität Bonn, Römerstr. 164, 53117 Bonn

zu kurz. Dafür gibt es eine Reihe von lernpsychologischen und fachlichen Gründen, auf die hier nicht gesondert eingegangen werden kann. Denn es ist das Anliegen dieses Aufsatzes, die alltägliche Praxis des Informatikunterrichts, die das Implementieren im erheblichen Umfang beinhaltet, zu analysieren. Denn Beobachtungen der alltäglichen Praxis haben ergeben, dass dem Implementieren sehr viel Zeit eingeräumt wird, ohne dass die Lernenden dabei zusätzliche Kompetenzen erwerben. Mehr noch: In der Phase des Implementierens wird die ohnehin schon erhebliche Heterogenität größer, weil insbesondere viel Zeit dafür ver(sch)wendet wird, syntaktisch korrekten Code zu erzeugen, und dabei die nicht intrinsisch motivierten SchülerInnen sehr vorsichtig und langsam agieren. Darüber hinaus befassen diese sich außerhalb des Unterrichts kaum bis gar nicht mit den Inhalten. Diese Unterschiede scheinen durch den nun vollzogenen Wechsel zur Objektorientierung größer geworden zu sein.

Im Folgenden wird die dahinter stehende komplexe Gemengelage in (untersuchbare) Teilaspekte zerlegt. Dies geschieht, wie in der Einleitung schon betont, in Form von Hypothesen, die anhand von Symptomen oder Befunden aus dem Informatikunterricht hergeleitet werden. Dazu wird zunächst dem Aspekt der Objektorientierung nachgegangen, von dem man sich einen einfacheren Zugang versprochen hatte, weil sie „der natürlichen Denkweise am nächsten kommt“ [Sc95, S. 186] und der möglichst dem imperativen Paradigma vorangehen sollte [Di07]. In diesem Zusammenhang hat es in den letzten Jahren eine ganze Reihe von Untersuchungen und Publikationen gegeben, die den Nutzen der Objektorientierung in ihrer Ambivalenz diskutieren (2.1). Danach wird auf populäre Entwicklungsumgebungen für den Informatikunterricht eingegangen, die u. a. das Ziel haben, den vielleicht zunächst nicht so motivierten SchülerInnen den Zugang zum Programmieren einfacher bzw. überhaupt schmackhaft zu machen. Es zeigt sich, dass diese Herausforderungen mit Technik (allein) nicht zu lösen sind (2.2). Zusätzlich wird auf grundsätzliche Schwierigkeiten des Informatik-Anfangsunterrichts hingewiesen, die sich daraus ergeben, dass man systematisch in einige Bausteine algorithmischer Sprache einführt (2.3). Ein Zwischenfazit beendet diesen Abschnitt (2.4).

2.1 Objektorientierung im Anfangsunterricht

Im Jahr 2009 erschien ein Aufsatz von Kortenkamp, Modrow, Oldenburg, Poloczek und Rabel [KM09], in dem sie sich kritisch mit der Objektorientierung im Informatikunterricht auseinandersetzen. Er endet mit diesem Satz:

„Die geringe Zahl von SchülerInnen, die Informatik in der Qualifikationsphase belegt, ist möglicherweise nicht nur den Zwängen des Zentralabiturs geschuldet, sondern auch eine Reaktion darauf, dass der Unterricht (immer noch) nicht an den Bedürfnissen der SchülerInnen orientiert ist.“ [KM09, S. 47]

Sie argumentieren in dem Aufsatz, dass OO nicht nur an den Bedürfnissen der SchülerInnen vorbeigeht, sondern auch an denen der Fachwissenschaftler, da auch die Vorbereitung auf ein Hochschulstudium nur unzureichend geleistet wird. Letzteres ist ein Aspekt, auf den Börstler schon zwei Jahre zuvor hingewiesen hatte [Bö07]. Außerdem schreiben

Kortenkamp und anderen:

„Die Behandlung von OOM ohne OOP steht vor einem Problem, das dem der Mengenlehre in der „Neuen Mathematik“ in den Siebziger Jahren des vorigen Jahrhunderts ähnelt: Die SchülerInnen lernen neue Wörter zur Beschreibung von Dingen, die sie schon kennen; sie erschließen sich damit aber keinen neuen Möglichkeiten, etwas zu tun oder zu verstehen.“ [KM09, S. 43]

Diese Analogie zur Mengenlehre erscheint insofern statthaft, da es vielen SchülerInnen in der Tat nicht gelingt, die Informatikbegriffe Klasse, Unterklasse, Objekt, Attribute und Attributwerte zu definieren und sinnvoll in ihre unterrichtlichen Äußerungen einzubauen. Dies scheint auch Folge des Zugangs; dies ist zugleich Inhalt von *Hypothese 1*: Es wird versucht, ohne Anschauung an einem Programm nur durch Bezug auf Fahrzeuge oder Tiere diese Begriffe (insbesondere unter Einbeziehung der Vererbung) vorab zu klären. Dass eine Klasse Teil des Programm-Codes ist, der diesen strukturiert und dass die Objekte (vielleicht sollte man besser von *Instanzen* sprechen) zur Laufzeit erzeugt werden, wird dabei zu wenig deutlich hervorgehoben. Insbesondere wird kaum darauf eingegangen, wie der Programm-Code im Computer verarbeitet wird.

Viele, die in der Objektorientierung eine Hoffnung auf einen Informatikunterricht gesehen haben, der besser zugänglich ist, beziehen sich auf einen Aufsatz von Schwill, der 1995 Programmierstile im Anfangsunterricht analysiert hat. Er schlussfolgert:

„Wir favorisieren für die Einführung in die Informatik den objektorientierten Ansatz vor allem aus drei Gründen: Erstens erfüllt dieses Paradigma unbestrittenermaßen informatisch orientierte Forderungen nach zeitgemäßem Unterricht mit mächtigen Konzepten … Zweitens ist dieser Ansatz im Sinne des didaktischen Prinzips der Fortsetzbarkeit, ein zentrales Merkmal eines nach dem Spiralpinzip organisierten Curriculums, auf höheren Niveau beliebig ausbaufähig. … Drittens – dies scheint aus pädagogischer Sicht der wichtigste Pluspunkt – ordnet sich der objektorientierte Stil in besonderer Weise harmonisch den elementaren kognitiven Prozessen unter, die beim Denken, Erkennen und Problemlösen im menschlichen Gehirn ablaufen.“ [Sc95, S. 183]

Letztlich beruht seine dritte Schlussfolgerung darauf, dass er ein kognitionspsychologisches Experiment zur Problemlösung (von Duncker aus den 1930er Jahren) einer Deutung zuführt, die Aspekte der Objektorientierung enthält. Einer dieser Aspekte, den Duncker *funktionale Bindung* nennt, behindert jedoch den Problemlöseprozess. Diese muss durch eine alternative Sicht auf die Objekte, durch eine andere *funktionale Bindung*, im Problemlöseprozess überwunden werden. Damit erscheint die Schlussfolgerung weit weniger zwingend, da objektorientiertes Denken den Widerspruch, dass die *funktionale Bindung* den Problemlöseprozess ggf. behindert, eigentlich nicht auflösen kann.

Die Vorteile der Objektorientierung ergeben sich, den Beobachtungen im Informatikunterricht folgend, vielmehr aus einem anderen Umstand. Dies ist dann *Hypothese 2:* Den SchülerInnen fällt es leichter die Objekte und deren Aufgaben in einem Anwendungskontext zu benennen und zu beschreiben als von Beginn an Algorithmen zu beschreiben. Damit wird die Zerlegung der Aufgabenstellung 'einfacher', als wenn sie direkt algorithmisch z. B. durch *stepwise refinement* erfolgen würde. Um dieses zu erreichen, müssen die Beispiele jedoch sinnvoll gewählt werden ('Hello World' oder mathematische

Aufgaben sind dann wohl kein sinnvoller Einstieg).

Zugleich muss man sich eingestehen, dass Algorithmen und imperative Programmierung nicht obsolet sind. Die Einführung der Objektorientierung hat dazu geführt, dass die SchülerInnen ein Mehr an Begriffen und Konzepten lernen müssen. Denn Objektorientierung ist – so wie sie in den Schulen am Beispiel der Sprache Java unterrichtet wird – vor allem eine *imperative Programmierung++*.[2] Aus dieser Erkenntnis ergibt sich ein pragmatischerer Zugang wie er 2008 von Hu vorgeschlagen wurde. Er überschreibt seinen Aufsatz mit „Just say 'a class defines a Data Type'" [Hu08]. Objektorientierung wird in diesem Sinne eine andere, vielleicht sogar elegantere bzw. eine größere Vielfalt zulassende Methode, Datentypen zu strukturieren und miteinander in Beziehung zu setzen.

Insofern ergibt sich *Hypothese 3*: Die Vorteile von OOM und OOP werden wohl insbesondere dann erkennbar, wenn man tatsächlich a) Vererbung benötigt wird, da durch eine Klassenhierarchie wie z. B. im Bereich der Komponenten für GUIs ein erheblicher Rationalisierungseffekt eintritt, b) die Algorithmen als Methoden sehr viel enger und auch in kompakterer Schreibweise (die Punktschreibweise) an die Datenobjekte gekoppelt werden können sowie c) in komplexeren Kontexten, an denen eine Vielzahl[3] von Objekten beteiligt ist und dadurch das Problem/die Aufgabe zerlegt werden kann.

Damit ist die ideologisch geführte Diskussion um „Objects later" bzw. „Objects first" für die Praxis weniger interessant. Vor allem die Untersuchungen von Ehlert (zum Teil mit Schulte) zeigen, dass letztlich keine Präferenzen angegeben werden können (vgl. hierzu die Zusammenfassung in [Eh12, S. 184ff]) Man kann zwar beobachten, dass Objektorientierung einen anderen Zugang zur Problemlösung erfordert als einen rein imperativen. Letztlich werden auch bei der Nutzung objektorientierter Sprachen Algorithmen entworfen. Dies verweist dann auf die *Hypothese 4*: In der imperativen Modellierung und Programmierung sozialisierte Personen lassen sich deswegen weniger auf objektorientiertes Modellieren ein, weil sie darum wissen, dass am Ende doch noch algorithmisiert werden muss. Diese durchaus schwierige Arbeit wollen sie so schnell wie möglich erledigen. Außerdem vermuten sie in der Suche nach Objekten und der Zuordnung von Aufgaben einen Overhead, der bei kleinen und bereits sehr gut mathematisierten bzw. algorithmisierten Kontexten, die oftmals für den Einstieg genutzt werden, auch vorhanden ist.

2.2 Programmierumgebungen für Anfänger

In den letzten 30 Jahren sind immer wieder neue Programmierumgebungen für den Informatik(anfangs)unterricht vorgestellt worden. Oftmals werden den SchülerInnen grafische Objekte zur Verfügung gestellt, die vorprogrammierte Funktionen enthalten, die dann weiter genutzt werden sollen. Anhand von zwei Beispielen wird im Folgenden kurz erläutert, dass hierdurch die Lernprozesse zum einen in Bezug auf die Mensch-Maschine-Interaktion erleichtert werden sollen und zum anderen Motivation geschaffen

[2] Vgl. C und C++. M. a. W.: Der Umfang des Lernstoffes wurde erhöht.
[3] Für Anfänger erscheinen drei unterschiedliche Objekte eine Vielzahl, die aber zugleich erschließbar ist.

werden soll, sich überhaupt mit dem Programmieren zu befassen. *Hypothese 5*: Die Vielzahl solcher Umgebungen scheint ein Indiz für den Bedarf zu sein und für die weiterhin vorhandene Unzufriedenheit mit den bisherigen Umgebungen. Hier gibt es Prozesse, die man technisch unterstützt haben möchte, ggf. sind diese Wünsche auf didaktischen Kontext bezogen. Die Anforderungen sind im Anfangsunterricht andere als später. Daher wird immer wieder versucht eine „eierlegende Wollmilchsau" zu erstellen.

Die beiden Beispiele, die im Folgenden näher betrachtet werden, sind *Alice*, das kaum eine Rolle (mehr) spielt und zum anderen *Greenfoot*, das derzeit auch unterstützt durch ein in 2.3 zu thematisierendes Schulbuch gerade eine große Rolle spielt.

Zu Alice: Pausch ist sozusagen der Vater von Alice. Da man darum wusste, dass er nicht mehr lange zu leben hat, wurde ihm die Gelegenheit gegeben, eine *Last Lecture* zu halten, in der er u. a. auf Alice einging. Sein Ziel war es, die Lernenden zu überlisten.

„So Alice is a project that we worked on for a long, long time. It's a novel way to teach computer programming. Kids make movies and games. The head fake, again, we're back to the head fakes. The best way to teach somebody something is to have them think they're learning something else. … the head fake here is that they're learning to program but they just think they're making movies and video games." [4]

Ein solcher *head fake* kann nur ein kurzfristig wirksamer Lernanreiz sein. Da Alice zudem nur vordergründig objektorientiert ist, sollte man sich an ihr nicht weiter orientieren (vgl. hierzu auch [DM09, En11]). In Alice (ähnlich wie z. B. in *Scratch*) wird das *Coden* durch Bau- bzw. Puzzlesteine unterstützt, sodass man schneller zu syntaktisch korrektem Code kommt. Dies schafft zwar ein wenig Raum für Kreativität, die jedoch mit den typischen Vorgehensweisen im Informatikunterricht im Konflikt steht (siehe 2.3).

Greenfoot ist hingegen eine Plattform, die in vielen Kursen (erfolgreich) genutzt wird. Dieser Aufsatz auf *BlueJ*, der auch von Kölling und Co entwickelt wurde, setzt letztlich den Ansatz von Mini-Welten fort, wie er durch *Logo*-Turtle, durch *Karol, the robot*, den Käfer *Kara* und viele mehr propagiert wird. In *Greenfoot* sind viele Implementationen in den Basisklassen *World* und *Actor* versteckt, wodurch die dahinterstehende Komplexität, mit der Anfänger auch nicht konfrontiert werden sollten, auch nicht erkennbar ist. Diese miteinzubeziehen würde wohl auch dazu führen, dass die SchülerInnen demotiviert werden. Die Implementation der Basisklassen der Szenarien (Roboter, Rover, Spinnen etc.) sind jedoch nicht versteckt, was im unterrichtlichen Geschehen u. a. dazu führen kann, dass gerade von pfiffigeren SchülerInnen die Szenarien manipuliert werden und korrekte Aufgabenlösungen nicht funktionieren. Zudem muss man *Greenfoot* irgendwann verlassen und auf eine andere Umgebung wechseln. Hier zeigt sich, wie sehr die gewonnenen Kenntnisse an die Umgebung gekoppelt sind und in andere Umgebungen, selbst wenn man nur zu *BlueJ* wechselt, nicht übertragen werden können (vgl. hierzu [En13]).

Hypothese 6: Der Versuch, mit Technik Motivation zu stiften, hat seine Grenzen. Letztlich werden die SchülerInnen mit nicht sonderlich kognitiv anspruchsvollen Aufgaben

[4] Vgl. z.B. http://theoncologist.alphamedpress.org/content/12/11/1374?trendmd-shared=1

(komplexerer) objektorientierter und algorithmischer Modellierungen und Implementierungen konfrontiert, deren direkter Sinn sich nicht jedem erschließt. Der Sinn dieser Vorgehensweise liegt wohl insbesondere darin, in die Elemente einer Programmiersprache einzuführen, was sich auch im Folgenden zeigt.

2.3 Typische Vorgehenswiesen im Anfangsunterricht

Greenfoot wird derzeit von vielen benutzt, da es anders als seine Basis *BlueJ* Anfänger besser unterstützt. Für NRW sind zur Nutzung von *Greenfoot* Schulbücher (z. B. [KL14] für die Einführungsphase) erschienen, die von vielen genutzt werden. Hier kann und soll keine ins Detail gehende kritische Bestandsaufnahme dieses Werkes erfolgen. Es soll nur kurz darauf hingewiesen werden, dass das Vorgehen in diesem Buch, wie zuvor auch im Einführungsband von Schriek [Sc05] („Stifte und Mäuse“, ähnlich wie das in NRW aktuell diskutierte *GLOOP*, ein grafischer Aufsatz auf *BlueJ*) oder auch anderen Büchern, die Reihenfolge der Sachlogik der Programmiersprache folgt. Die Inhalte Blockstrukturen, Schleifen, Verzweigungen, Variablen, Datenstrukturen werden in eine sachlogisch korrekte Reihenfolge gebracht. *Hypothese 7*: Im Informatikunterricht steht weniger das Erlernen bzw. Einüben des algorithmischen Problemlösens oder von informatischem Denken, vielmehr das Erlernen einer Programmiersprache im Vordergrund. Die SchülerInnen lösen nicht wirklich Probleme, sondern nur Aufgaben, die in ihrer Reihenfolge der Logik des Erlernens der Programmiersprache untergeordnet sind. Die Zusammenhänge zum Problemlösen werden nicht erkannt. Denning, Tedre und Youngpradit haben erst kürzlich in einem 'Viewpoint‘ in der CACM darauf hingewiesen, dass dieser Transfer bisher lediglich behauptet, aber nie nachgewiesen wurde. [DT17]

Außerdem müssen die SchülerInnen konstruierend vorgehen. Dies hat sicher lernpsychologische Vorteile, hat aber mit Blick auf die zu verwendenden Instrumente (die Entwicklungsumgebungen) auch Nachteile, da nicht zu hundert Prozent syntaktisch korrekte Lösungen zurückgewiesen werden. Aber selbst wenn das nicht geschieht, kommt es auch weiterhin zu Fehlern zur Laufzeit (*Stack overflow*, *Null pointer exception* …) bzw. zu unerwünschten Systemverhalten, deren Korrektur nicht nur ein profundes Verständnis der internen Datenverarbeitungsprozesse erfordern, das (s. o.) nicht unbedingt zuvor entwickelt wurde. *Hypothese 8:* Dieser Wechsel zwischen den Ebenen ist eine Abstraktionsfähigkeiten abnötigende kognitive Leistung, die sich wiederum in einem Kontext stattfindet, der nur dem Erlernen einer Programmiersprache und weniger dem Erwerb von Problemlösekompetenzen dient. Die SchülerInnen werden lediglich gefordert, aber nicht notwendigerweise gefördert.

2.4 Zwischenfazit

Programmieren Lernen ist nicht nur der heimliche, sondern der offensichtliche Lehrplan zu Beginn der Einführungsphase. Darüber hinaus gehende Lernziele werden kaum erreicht. Dieser Lehrplan soll durch didaktische Tricks oder bunte Animationen schmack-

haft oder besser zugänglich gemacht werden. Diese Effekte sind jedoch nicht nachhaltig, auch weil es *nicht* das Bedürfnis der allermeisten SchülerInnen zu sein scheint, Programmieren lernen zu wollen. Dass sie dabei auch konstruktiv tätig werden müssen, aber von dem Instrument, mit dem sie arbeiten, immer wieder zurecht oder garzurückgewiesen werden, trägt nicht zur Motivation bei; im Gegenteil! Erfolgserlebnisse und Sinnstiftung fehlen gleichermaßen. Zumindest eine solche Sinnstiftung wird durch den im Folgenden darzustellenden Zugang versucht.

3 Ein alternativer Zugang

Vergleicht man diese Schwierigkeiten des Informatikunterrichts mit anderen Fächern, ist der konstruktive Zugang ungewöhnlich. In vielen anderen Unterrichtsfächern steht eher die Analyse von Gegenständen, Phänomenen und Situationen der Natur, der Kultur oder der Zivilisation im Vordergrund. Im naturwissenschaftlichen Unterricht setzt man dabei im besten Fall auf entdeckendes Lernen in Form von Experimenten. Auch im Musikunterricht wird eher analysiert denn komponiert, ggf. wird musiziert, aber auch das wohl seltener, da man hier auf Probleme der Vorbildung und der starken Differenzierung stößt (Begabung).[5] Aber selbst, wenn man im Fach Musik 'konstruktiv' vorgeht; die Instrumente würden den Versuch nicht zurückweisen. Allenfalls sind Disharmonien vernehmbar. Im Kunst- und – so es ihn gibt – im Technikunterricht wird konstruktiv gearbeitet, allerdings auch dort ohne grundsätzliche Zurückweisungen durch die Instrumente. Also eher rekonstruktiv bzw. analytisch vorzugehen, ist eine Facette dieses Zugangs.

Die andere Idee ist, die Aufgaben in ein Software-Projekt einzubetten. Das ist nicht neu und orientiert sich grob an den Vorstellungen von Magenheim [Ma08]. Zugleich wird der Versuch unternommen, diesen hochkomplexen und bislang nicht wirklich operationalisierten Ansatz für den Anfangsunterricht zu adaptieren. Verschiedene Versionen bzw. Entwicklungsstufen einer Software werden zyklisch analysiert und dann weiterentwickelt. Es wird mit der Analyse eines sehr einfachen und kurzen Programm-Codes gestartet. Es werden mithin zuerst die Bestandteile im Programm-Code aufgespürt und mit den korrekten Begriffen benannt, ähnlich wie man es Fach Deutsch mit Prosa, Lyrik aber auch Sachtexten macht, wobei dort der Vokabelapparat ein viel größerer ist. Erst nach Klärung der Begriffe, Konzepte und der Verarbeitungsmechanismen wird dann selbst implementiert.

Als Kontext für das Software-Projekt dient ein Brettspiel. Die Befragung von Rabel und Oldenburg [RO09] verweist darauf, dass dies durchaus an die Bedürfnisse der SchülerInnen anschließt, obschon die sich sicher eine hippere Spieleentwicklung wünschen würden.[6] Am Beispiel des Spiels Kniffel (Knobeln, Yathzee) wird im Folgenden dieser Ansatz konkretisiert, wobei Kniffel nur ein Beispiel unter vielen möglichen ist. Hier erscheint Zugänglichkeit für die SchülerInnen (einfache Regeln, wenige aber unter-

[5] … auch eine Analogie zur Informatik?

[6] . Auch Diethelm konkretisiert ihre Vorstellungen an einem Gesellschaftsspiel [Di07].

scheidbare Objekte, die zum Teil baugleich sind) jedoch in besonderer Weise gegeben.

Man startet zunächst mit der Implementation eines Würfels, der als Klasse (als Programm-Code und als UML-Klassendiagramm) den SchülerInnen gegeben wird. Code wie auch Diagramm werden auf ihre Bestandteile hin analysiert. Diese für jede Klasse gleiche Struktur wird ausführlich erarbeitet und die Beziehungen zwischen Code und Klassendiagramm können entdeckt werden. Der Code wird instanziiert und damit kann der Würfel auch „gerollt“ werden. Da die Klasse eine Grafikkomponente enthält (hiermit werden also auch Ideen der Mini-Welten aufgegriffen) sehen die SchülerInnen, dass etwas passiert und auch was. An dieser Stelle sind Experimente mit dem Code (nicht nur Sechser-Würfel, andere GUI …) erwünscht. In dieser Phase sollte man sich Zeit lassen, die Beziehungen von Code und Programm mit den dazugehörigen Begrifflichkeiten zu klären, ohne dass man den SchülerInnen auch noch ein konstruktives Vorgehen abverlangt. Die Begriffe Objekt, Klasse etc. werden in ihren informatischen Kontext gestellt.

In einem zweiten Schritt soll der Übergang zum Würfelbecher stattfinden, der auf der Grundlage der Erkenntnisse der ersten Phasen von den SchülerInnen selbst modelliert und dann implementiert werden soll. An dieser Stelle können verschiedene Varianten der 'Kooperation' der Objekte Becher und Würfel diskutiert werden. Die SchülerInnen werden erkennen, dass die Klasse Würfel erweitert werden muss, um den wechselseitigen Bezug von Würfel und Becher zu implementieren. Am Ende dieser zweiten Phase, wenn die SchülerInnen den Würfelbecher implementiert haben, der dann nur die Würfel rollt, die auch im Würfelbecher sind, können sie bereits viele der z. B. in dem Lehrplan NRW genannten Kompetenzen erworben haben, da sie Teil des Unterrichts waren.[7]

Es wäre dann zu überprüfen, inwieweit diese Kompetenzen entwickelt wurden. Außerdem wäre zu untersuchen, wie SchülerInnen mit Vorkenntnissen mit diesem Zugang umgehen und wie motiviertere und begabtere SchülerInnen mit der Intensität des zunächst analytischen Vorgehens umgehen. Das Szenario scheint jedoch genügend viele Möglichkeiten zur Binnendifferenzierung zu bieten, bei der sich die 'Besseren' z. B. schon der Auswertung von Würfelergebnissen (ist es eine *Straße*, *Full House* …) widmen oder einen digitalen Notizzettel erstellen (der alle Einträge zulässt, wie sein Vorbild aus Papier oder nur korrekte bzw. mit dem Würfelergebnis konsistente). Als besondere Herausforderung könnte ein „Computerspieler“ implementiert werden.

4 Alltagspraxis: Ein Forschungsansatz zur Prozessbeobachtung

Es wurde schon mehrfach betont, dass es hier nur Hypothesen (begründet oder plausibel) formuliert wurden und dass auch der gerade vorgestellte alternative Ansatz noch der Evaluation bedarf. Evaluation bedeutet – wie gerade ausgeführt – Kompetenzmessung, aber nicht nur. In dem hier vorzustellenden Ansatz der Alltagspraxis geht es um die

[7] Diese können hier aus Platzgründen nicht aufgezählt werden. Vgl. http://www.schulentwicklung.nrw.de/lehrplaene/upload/klp_SII/if/KLP_GOSt_Informatik.pdf S. 21-23

Erforschung der (alltäglichen) Prozesse im Unterricht, für die viele der Hypothesen noch genauer zu Forschungsfragen zu verdichten wären. Dazu gehört die Beobachtung der Aktivität der SchülerInnen (der Unterrichtsprozesse) als wichtige zu ergänzende Komponente. Da einige der Tätigkeiten an bzw. mit den Rechnern erfolgen, könnten via Eye-Tracking oder der Aufzeichnung von Tastatur und Mausevents Daten erhoben werden, die zwar keine absolute Bedeutung haben, aber möglicherweise durch begleitende qualitative Forschung mit relativen Bedeutungen versehen werden können. Zu diesem Ansatz gehören auch freie Beobachtungen, in denen man versucht weitere Phänomene aufzuspüren und dann in anderen Beobachtungen mit anderen Personen wiederzuentdecken (ethnografische Studien). Begleitet werden sollte das Ganze durch Interviews mit LehrerInnen sowie SchülerInnen, die dann qualitativ analysiert werden und mit deren Ergebnissen die Hypothesen und Forschungsfragen weiter präzisiert werden.

5 Fazit mit Ausblick

In diesem Aufsatz wurden auf der Grundlage von einer ganzen Reihe von Beobachtungen zum Informatikunterricht – auch im Vergleich zu anderen Fächern – Hypothesen zu den besonderen didaktischen Herausforderungen des Informatikunterrichts aufgestellt. Dazu wurde ein Ansatz vorgestellt, der nun evaluiert werden kann. Eine möglichst große Anzahl von LehrerInnen sowie SchülerInnen sollten damit konfrontiert werden. Deren Umsetzungen sollten mit den Prozessen und Erträgen konventionellen Informatikunterrichts verglichen werden, sofern sie vergleichbar sind. Alle anderen Informatik-DidaktikerInnen werden hiermit aufgefordert, sich an diesen Studien zu beteiligen.

Insgesamt wird damit ein empirischer Forschungsansatz vorgelegt, der dazu dienen soll, die durchaus widersprüchlichen Phänomene (sehr motivierte und ganz wenig motivierte SchülerInnen, große sowie ausbleibende Lernerfolge …) zu systematisieren und über alternative Vorgehensweisen nachzudenken. Auch weitere neuralgische Stellen, wie z. B. die Behandlung von Algorithmen mit (dynamischen) Datenstrukturen erfordern einen eher analytisches und weniger konstruktives (= modellierendes und implementierendes) Vorgehen. Der Platz in diesem Beitrag reicht dafür nicht aus, dieses darzustellen.

6 Literatur

[Bö07] Börstler, J.: Objektorientiertes Programmieren – Machen wir irgendwas falsch? In: Sigrid Schubert (Hrsg.): Didaktik der Informatik in Theorie und Praxis. INFOS 2007, 12. GI-Fachtagung Informatik und Schule GI-Edition - Lecture Notes in Informatics (LNI), P-112. Bonner Köllen Verlag (2007)

[Di07] Diethelm, I.: "Strictly models and objects first" - Unterrichtskonzept und -methodik für objektorientierte Modellierung im Informatikunterricht, Dissertation, Universität Kassel, Fachbereich Elektrotechnik/Informatik, 2007 http://kobra.bibliothek.uni-kassel.de/bitstream/urn:nbn:de:hebis:34-2007101119340/1/DissIraDruckfassungA5.1.pdf

[DM09] Dohmen, M., Magenheim, J., Engbring, D.: Kreativer Einstieg in die Programmierung - Alice im Informatik-Anfangsunterricht. In: Peters, I. (Hrsg.): Informatische Bildung in Theorie und Praxis, Beiträge zur INFOS 2009, 13. GI-Fachtagung - Informatik und Schule, S.69-80, Berlin (LOG IN Verlag) 2009

[DT17] Denning, P.J.; Tedre, M.; Youngpradit, P.: The Profession of IT. Misconception about Computer Science. Communications of the ACM. March 2017/Vol. 60, No. 3, S. 31 – 33. doi:10.1145/3041047

[Eh12] Ehlert, A.: Empirische Studie: Unterschiede im Lernerfolg und Unterschiede im subjektiven Erleben des Unterrichts von Schülerinnen und Schülern im Informatik-Anfangsunterricht (11. Klasse Berufliches Gymnasium) in Abhängigkeit von der zeitlichen Reihenfolge der Themen (OOP-First und OOP-Later), Dissertation, FU Berlin, 2012 http://www.diss.fu-berlin.de/diss/receive/FUDISS_thesis_000000035764

[En11] Engbring, D.: Untersuchungen und Bewertungen zum Einsatz von Alice im Informatikunterricht. In: M. Weigend, M. Thomas, F. Otte (Hrsg.): Informatik mit Kopf, Herz und Hand. Praxisbeiträge zur INFOS 2011. ZfL-Verlag. Münster, S. 81 - 90

[En13] Erst nehmen wir Greenfoot. Und dann BlueJ? In: Breier, N., Stechert, P., Wilke, T. (Hg.): INFOS 2013. 15. GI-Fachtagung Informatik und Schule. Praxisband. Kiel Computer Science Series. 2013/3. S. 29 – 39

[Hu08] Hu, C.: Just say 'A class defines a Data Type'. In: Communications of the ACM, Vol. 51 No. 3, Pages 19-21 doi: 10.1145/1325555.1325560

[KL14] Kempe, T.; Löhr, A.: Informatik 1. Schoeningh-Verlag. Paderborn. 2014

[KM09] Kortenkamp, U; Modrow, E.; Oldenburg, R.; Poloczek, J; Rabel, M.: Objektorientierte Modellierung - aber wann und wie? Zur Bedeutung der OOM im Informatikunterricht. LOG IN 160/161, 2009. S. 41-47

[Ma08] Magenheim, J.: Systemorientierte Didaktik der Informatik Sozio-technische Informatiksysteme als Unterrichtsgegenstand? In Kortenkamp; U.; Weigand; H.G.; Weth, T. (Hrsg.): Informatische Ideen im Mathematikunterricht, Franzbecker Hildesheim 2008. S. 17 - 36

[RO09] Rabel, M; Oldenburg, R.:Konzepte, Modelle und Projekte im Informatikunterricht – Bewertungen und Erwartungen von Schülern und Studenten. In: Bernhard Koerber (Hrsg.): Zukunft braucht Herkunft. 25 Jahre »INFOS – Informatik und Schule« INFOS 2009, 13. GI-Fachtagung »Informatik und Schule« GI-Edition - Lecture Notes in Informatics (LNI), P-156. Bonner Köllen Verlag (2009), S. 146 -155

[Sc95] Schwill, A.: Programmierstile im Anfangsunterricht. In: S. Schubert (Hrsg.).: Innovative Konzepte für die Ausbildung 6. GI-Fachtagung Informatik und Schule. Springer. Berlin Heidelberg. S. 178-187

[Sc05] Schriek, B.: Informatik mit Java: Eine Einführung mit BlueJ und der Bibliothek Stifte und Mäuse. Band 1. Nili-Verlag. Werl

[Wi06] Wing, J.: Computational Thinking. Communications of the ACM. March 2006/ Vol. 49, No. 3, 33-35